CSC263 Assignment #1

Most of algorithm questions have 5 points: 3 points for pseudocode, 1 point for justifying correctness, and 1 point for running time analysis.

Each "small" error (missing or wrong boundary conditions, missing return values, uninitialized variables, etc) in pseudocode deducts 1 point, but no more than 2 points for such errors. As long as the main idea of the code is correct, it should get at least 1 point.

1.

(a)

(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)

For indices starting from 0:

(0, 4), (1, 4), (2, 3), (2, 4), (3, 4)

(b)

The array with all elements reversely sorted, that is, $[n, n\text{-}1, \ldots, 2, 1]$.

Since every pair of distinct element is an inversion, the number of inversions is $n(n\text{-}1)/2$.

(c)

Let M be the number of inversions. The running time is $O(n + M)$.

The outer loop of insertion sort runs $O(n)$ times. The inner loop runs $O(M)$ times in total.

Another solution is $O(\max\{n, M\})$, which is equivalent to $O(n + M)$.

Note $O(M)$ is wrong since it could happen that $M < n$.

(d)

We give an algorithm MergeSortWithInversion which extends merge sort to count the number of inversions. The algorithm divides the input array in two parts, recursively sorts both parts, and counts the number of inversions which has one element in the first half and the other in the second half.

```
MergeSortWithInversion($A, p, r$)
    $cnt = 0$
    **if** $p < r$
        $q = \lfloor (p + r) / 2 \rfloor$
        $cnt$ += MergeSortWithInversion($A, p, q$)
        $cnt$ += MergeSortWithInversion($A, q+1, r$)
        $cnt$ += MergeWithInversion($A, p, q, r$)
    **return** $cnt$

MergeWithInversion($A, p, q, r$)
    $cnt = 0$
    $n_1 = q - p + 1$
    $n_2 = r - q$
```

```
copy $A[p, q]$ to $L[1..n_1]$
copy $A[q+1, r]$ to $R[1..n_2]$
$L[n_1+1] = R[n_2+1] = \infty$
$i = j = 1$
for $k = p$ to $r$
    if $L[i] \le R[j]$
        $A[k] = L[i]$
        $i$++
    else
        $A[k] = R[j]$
        $j$++
        $cnt$ += $n_1 - i + 1$                // A common mistake is $cnt$ += $n_1 - i$
return $cnt$
```

To run the algorithm we need call MergeSortWithInversion(A, 1, $A.length$).

The MergeWithInversion method extends the Merge method of merge sort by counting the number of inversions between the parts. Since both the left part L and the right part R are sorted, whenever we find a smaller element from R, it has inversions with each element in $L[i..n_1]$. Each inversion (a, b) is counted at some call of MergeWithInversion when the two parts a and b belong to get merged, and counted only once since after merging the inversion does not exist any more.

The running time is $\Theta(n \log(n))$. The analysis is the same as for merge sort. We recursively divide the array, and start merging when the subarrays have size 1. There are $\Theta(\log n)$ levels of divisions, and merging all pairs of subarrays at each level takes time $\Theta(n)$. Thus the total running time is $\Theta(n \log(n))$.

2.

(a)

```
ExtractSecondLargest($A$)
    if $A.heapsize < 2$
        then error 'Heap has only 1 element'
    $i = 2$
    if $A.heapsize > 2$ and $A[3] > A[2]$
        $i = 3$
    $m = A[i]$
    $A[i] = A[A.heapsize]$
    $A.heapsize = A.heapsize - 1$
    MaxHeapify($A$, $i$)
    return $m$
```

Another solution is to call ExtractMax twice, and then insert the largest one back. This is also correct, although it is not a good one.

Although not a good solution, I gave this solution full marks (5pts) as well as it satisfies the O(logN) requirement from the question.

(b)
One of $A[2]$ and $A[3]$ must be the second largest since they are the largest in the left and right subtrees of the root, respectively. Moreover, the last element must be no larger than $A[i]$, so after we use it to replace $A[i]$, we can use MaxHeapify to "buble down".

The running time is $O(\log n)$, since MaxHeapify takes time $O(\log n)$.

3.
(a)
Similar as in ExtractMax, we will use the last element in the heap to replace $A[i]$. If it is smaller than the last element, then we call IncreaseKey to replace it by the last element and "bubble up"; otherwise, we replace it by the last element and call MaxHeapify to "bubble down".

```
HeapDelete(A, i)
    if A.heapsize < i
        then error 'Heap does not have ith element'

    last = A[A.heapsize]
    A.heapsize = A.heapsize − 1
    if A[i] < last                  // bubble up
        IncreaseKey(A, i, last)
    else                            // bubble down
        A[i] = last
        MaxHeapify(A, i)
```

The last element could be larger or smaller than $A[i]$, thus there must be both "bubble up" and "bubble down". **Important note as it is a common mistake: This is because the last element may not necessarily be a descendent of node [i], so it could be bigger.**

Another solution I found is to just use IncreaseKey(A, i, >= maxValue), (there will be 2 max values, or a new max value assuming existing wasn't maximum possible value), then extractMax, which is also correct. I deducted 1pt if they did IncreaseKey(A, i, maxValue + 1), but gave full if they did IncreaseKey(A, i, maxValue) or IncreaseKey(A, i, infinity).

(b)
After replacing $A[i]$ by the last element and shrinking the heap size by 1, we either "bubble down" or "bubble up" it based on whether it is smaller than its parent. If the new $A[i]$ becomes larger, then it must be also larger than its children; we need to use IncreaseKey to continuously swap with the parents to get to the right position. If the new $A[i]$ becomes smaller, then the path from it to the root is good, but it might be smaller than its children; we use MaxHeapify to swap it down to the right position.

Since "bubble down" and "bubble up" both run in time linear in the height of the heap, we have HeapDelete runs in $O(\log n)$.

4.

(a)

Similar to binary heap, store the heap tree level by level and from left to right, with the root as the first element, its d children as the 2nd, 3rd, …, (d+1)-th elements, and so on.

For the ith element, its parent has index $\lfloor(i + d - 2)/d\rfloor = \lfloor(i - 2)/d\rfloor + 1$; its d children have indices $di - d + 2$, $di - d + 3$, …, $di + 1$.

When using indexing starting from 0, for the element with index i, its parent has index $\lfloor(i - 1)/d\rfloor$; its d children have indices $di + 1$, $di + 2$, …, $di + d$.

i starts from 0
parent[i] = floor((i-1)/d)
kthChild[i][k] = di + k

i starts from 1
parent[i] = floor((i-2)/d) + 1 = ceil((i-1)/d)
kthChild[i][k] = d(i-1) + 1 + k

(b)

The height is

$$\lfloor \log_d[(n-1)(d-1)+1] \rfloor \;=\; \lfloor \log_d(nd - n + 2) \rfloor - 1.$$

Or equivalently, the height is

$$\lceil \log_d[n(d-1)+1] \rceil - 1 = \lceil \log_d(nd - n + 1) \rceil - 1.$$

This is in $O(\log_d n)$.

A complete d-ary tree of height h has $1 + d + \ldots + d^h = (d^{h+1} - 1)/(d-1)$ elements. The smallest d-ary tree of height h has $1 + d + \ldots + d^{h-1} + 1 = (d^h - 1)/(d-1) + 1$ elements. Let n be the number of elements, and let

$$(d^h - 1)/(d-1) + 1 \;\le\; n \;\le\; (d^{h+1} - 1)/(d-1) \;<\; (d^{h+1} - 1)/(d-1) + 1.$$

We get

$$h \;\le\; \log_d[(n-1)(d-1)+1] \;<\; h + 1.$$

Or let

$$(d^h - 1)/(d-1) \;<\; (d^h - 1)/(d-1) + 1 \;\le\; n \;\le\; (d^{h+1} - 1)/(d-1).$$

We get

$$h < \log_d[n(d-1)+1] \;\le\; h + 1.$$

You must give the exact form and provide proofs to get full marks. Writing only the correct exact form gets 1 mark. Writing in wrong exact forms should get 0 marks, for example $\log_d n$, or $\lfloor \log_d n \rfloor$, or others.

(c)

```
ExtractMax($A$, $d$)
    if $A.heapsize < 1$
        error "Heap underflow"
    $max = A[1]$
    $A[1] = A[A.heapsize]$
    $A.heapsize = A.heapsize - 1$
    MaxHeapify($A$, $d$, 1)
    return $max$
```

```
MaxHeapify($A$, $d$, $i$)
    $largest = i$
    for $j = (di - d + 2)$ to $(di + 1)$
        if $j \le A.heapsize$ and $A[j] > A[largest]$
            $largest = j$
    if $largest \ne i$
        exchange $A[i]$, $A[largest]$
        MaxHeapify($A$, $d$, $largest$)
```

The running time is in $O(d \log_d n)$. The number of times MaxHeapify is called is bounded by the tree height $O(\log_d n)$, and each call itself takes $O(d)$.

(d)

```
IncreaseKey($A$, $d$, $i$, $key$)
    if $key < A[i]$
        error "new key is smaller than current key"
    $A[i] = key$
    while $i > 1$
        $j = \lfloor (i - 2) / d \rfloor + 1$
        if $A[j] \ge A[i]$
            break
        exchange $A[j]$, $A[i]$
        $i = j$
```

The running time is in $O(\log_d n)$, since the number of times the loop body runs is bounded by the tree height $O(\log_d n)$, and each takes $O(1)$.

(e)

```
Insert($A$, $key$)
    $A.heapsize$++
    $A[A.heapsize] = -\infty$
    IncreaseKey($A$, $d$, $heapsize$, $key$)
```

The running time is in $O(\log_d n)$, the same as IncreaseKey.

Note: You cannot leave d out from the complexity, it is NOT a constant, but a variable.

5.

(a)

15	9	8	5
14	4	1	$-\infty$
12	3	$-\infty$	$-\infty$
$-\infty$	$-\infty$	$-\infty$	$-\infty$

There are many other possible answers.

Needs to be somewhat upper left triangular.

(b)

By the property of reverse Young tableau, $Y[1, 1]$ is the largest element. If it is $-\infty$, every other element must be $-\infty$ too since they are smaller than or equal to $Y[1, 1]$.

By the property of reverse Young tableau, $Y[m, n]$ is the smallest element. If it is not $-\infty$, every other element must not be $-\infty$ since they are larger than or equal to $Y[m, n]$.

(c)

$Y[1, 1]$ is the largest element. We will replace it by $-\infty$, and then shift it the correct position by comparing with the elements below and on its right.

We define a method FixDown to recursively shift an element one position to the right or below based on its comparison with the two neighbors.

```
FixDown($Y$, $m$, $n$, $i$, $j$)
    $maxi = i$
    $maxj = j$
    if $i < m$ and $A[i + 1, j] > A[maxi, maxj]$
        $maxi = i + 1$
        $maxj = j$
    if $j < n$ and $A[i, j + 1] > A[maxi, maxj]$
        $maxi = i$
        $maxj = j + 1$
    if $maxi \neq i$ or $maxj \neq j$
        exchange $A[i, j]$ and $A[maxi, maxj]$
        FixDown($Y$, $m$, $n$, $maxi$, $maxj$)

ExtractMax($Y$, $m$, $n$)
```

```
    $max = Y[1, 1]$
    $Y[1, 1] = -\infty$
    FixDown($Y, m, n, 1, 1$)
    **return** $max$
```

The running time is $O(m + n)$. Each call of FixDown shift the new value $-\infty$ one position down or to its right; there are at most m times shifting down and at most n times shifting to the right. The total number of calls of FixDown is $m + n$. Each run of FixDown itself takes constant time; thus the total time is $O(m + n)$.

(d)
If Y is not full, then $Y[m, n]$ must be $-\infty$. We replace it by the inserted element, and then shift it the correct position by comparing with the elements above and on its left. The process is exactly the opposite of what we did for ExtractMax.

We define a method FixUp to recursively shift an element one position to the left or above based on its comparison with the two neighbors.

```
FixUp($Y, i, j$)
    $mini = i$
    $minj = j$
    if $i > 1$ and $A[i-1, j] < A[mini, minj]$
        $mini = i - 1$
        $minj = j$
    if $j > 1$ and $A[i, j-1] < A[mini, minj]$
        $mini = i$
        $minj = j - 1$
    if $mini \neq i$ or $minj \neq j$
        exchange $A[i, j]$ and $A[mini, minj]$
        FixUp($Y, mini, minj$)

Insert($Y, m, n, x$)
    $Y[m, n] = x$
    FixUp($Y, m, n$)
```

The running time is $O(m + n)$. The analysis is the same as in ExtractMax.

(e)
We will define a method to recursively eliminate one row or one column each time, until the target is found or all rows and columns are eliminated. The elimination is based on comparing the target with the element on the top-right corner. If the target is smaller, we eliminate the first row since all elements in it should be larger than the target; if the target is larger, we eliminate the last column since all elements in it should be smaller than the target.

```
Search($Y, i, j, m, n, x$)
```

```
        if $i > m$ or $j > n$
            return FALSE
        if $Y[i,\ n] == x$
            return TRUE
        if $Y[i,\ n] > x$
            $i = i + 1$
        else // $Y[i,\ n] < x$
            $n = n - 1$
        return Search($Y, i, j, m, n, x$)
```

We run on Search(Y, 1, 1, m, n, x) for the searching. The running time is $O(m + n)$, since in each call of Search, we reduce the search space by eliminating either one row or one column. Thus the total number of calls to Search is at most $m + n$, and the total running time is $O(m + n)$.

An iterative solution:

```
Search($Y, m, n, x$)
        $i = 1$
        $j = n$
        while $i \leq m$ and $j \geq 1$
            if $Y[i, j] == x$
                return TRUE
            if $Y[i, j] > x$
                $i$ ++
            else // $Y[i, j] < x$
                $j$ −−
        return FALSE
```

This solution traverses diagonally from top right to bottom left.
You could also traverse diagonally from bottom left to top right.

(f)
We will continuously call ExtractMax to until no element is left.

```
Sort($Y, m, n$)
        let $A$ be an array of length $mn$
        $size = 0$
        while TRUE
            $x$ = ExtractMax($Y, m, n$)
            if $x == -\infty$
                break
            $size$++
            $A[size] = x$
        for $i = size$ to 1
            print $A[i]$
```

For running Sort(Y, n, n), it takes time $O(n^3)$, since each call of ExtractMax(Y, n, n) takes time $O(n)$ and there are totally at most n^2 elements to sort.

6.

We will first use a subroutine to count the number of nodes n in the tree, and then continuously call Successor until $\lceil n / 2 \rceil$.

```
Count(x)
    if x == NIL
        return 0
    return Count(x.left) + Count(x.right) + 1

FindMedian(x)
    m = ⌈ Count(x) / 2 ⌉
    y = Minimum(x)
    for i = 2 to m
        y = Successor(y)
    return y
```

FindMedian must be an in-order traversal. The code is prolly wrong if it is pre-order or post-order traversal.

Starting from the minimum, k successive calls of Successor gives the $(k+1)$-th element. Thus, the returned is the $\lceil n / 2 \rceil$-th element, which is the median.

Count takes time $O(n)$. And $k < n$ successive calls of Successor also takes time $O(n)$. To see this, each edge of the tree is visited at most twice during all calls of Successor, once from the parent to the child, and once in the opposite way. There are $n - 1$ edges in total, so the time for k calls of Successor takes time $O(n)$. Therefore, the total running time is $O(n)$.

A cleaner solution but takes O(n) space is to return a sorted array of inorder traversal, then return the median of the sorted array.